Curso de Go(golang)

3.1. Hola Mundo

Para crear un hola mundo en Go podemos crear un archivo de `main.go`

- Primero tenemos que definir el paquete `main` que no permitirá ejecutar el archivo
- Para mostrar datos por pantalla o consola importar el paquete fmt
- Para ejecutar nuestro código se tiene que definir la función main y dentro de esta función
- Para mostrar un Hola mundo usamos el paquete fmt y su método Println

```go
package main

import "fmt"

func main() {

    fmt.Println("Hola Mundo")

}
```

Comentarios en Go: Los comentarios en lenguaje de programación sirven para comentar un línea de código y para documentar tu trabajo, como indicar que el siguiente bloque de código o línea de código y ubicarte fácilmente, estos comentarios son ignorados por el compilador de Go.

```go
package main //Definir el Paquete Principal

import "fmt" //Importar el paquete fmt

func main() { //Definir la función main

    /*

        Usar el paquete fmt y sus métodos

        para mostrar datos por pantalla

    */

    fmt.Println("Hola Mundo")

    fmt.Println("Empezando con Go")

}
```

3.2. Tipos de datos

Lenguaje de programación Go, es tipado entonces tenemos que conocer los tipos de datos par definir una variable.

- Números enteros
- Números Flotantes
- Cadena de Textos
- Booleanos - true y false
- Derivados - Punteros, arreglos, estructuras, uniones, funciones, slices, interfaces, Maps, Channels.

Rango de valores de números enteros

- uint8 - 8-bits (0 a 255)
- uint16 - 16-bits (0 a 65535)
- uint32 - 32-bits (0 a 4294967295)
- uint64 - 64-bits (0 a 18446744073709551615)
- int8 - 8-bits (-128 a 127)
- int16 - 16-bits (-32768 a 32767)
- int32 - 32-bits (-2147483648 a 2147483647)
- int64 - 64-bits (-9223372036854775808 a 9223372036854775807)

Rango de valores de números flotantes

- float32
- float64
- complex64
- complex128

Otra implementación de tipo de datos en Go.

- byte - 8-bits (0 a 255)
- rune - 32-bits (-2147483648 a 2147483647)
- uint - 32 o 64 bits
- int - 32 o 64 bits

```
//Cadena de caracteres
 fmt.Println("Hola") // string
 //Números Enteros
 fmt.Println(25) // int
 //Números Flotantes
 fmt.Println(2.5) // float64 o float32
 //Booleanos bool
 fmt.Println(true)
 fmt.Println(false)
```

3.3. Variables

Los variables son como un tipo de memoria temporal que se almacena en memoria RAM, en Go los variables son tipados a si que tenemos que definir un tipo de variable para un tipo de dado.

- Las variables de definen con la instrucción var luego el nombre o identificador de la variable y luego el tipo de dato.

```
//Definir una variable
var nombre string
```

- luego de definir puedes asignar un valor, mostrar los datos de la variable y también modificar el valor asignado.

```
nombre = "Alex" //Asignar un Valor
fmt.Println(nombre) //Mostrar el valor de la Variable
nombre = "Roel" //Modificar el valor de la Variable
fmt.Println(nombre)
```

- En Go podemos definir variables de muchas formas, como definir múltiples variables de un solo tipo.

```
//Definir variables múltiples de un solo tipo
 var a, b int = 1, 2
 var c, d int
 //Asignado valores
 c = 3
 d = 4
```

```
 //Mostrar por pantalla
 fmt.Println(a, b, c, d)
```

- También definir múltiples variables de varios tipos de datos.

```
//Definir variables múltiples de varios tipos
 var (
   pi      float64
   booleano bool
   cadena  = "texto 01"
   edad    = 25
 )
 pi = 3.141592
 booleano = true
 fmt.Println(pi, booleano, cadena, edad)
```

- Definir variables sin poner la instrucción var y sin indicar el tipo de dato, pero tenemos que asignar un valor y de acuerdo a su valor como los lenguajes dinámicos se convierte en tipo de variable.

```
 //Definir variable sin tipo de dato
 v1 := 24          // Int
 v2 := "Texto 02" // string
 fmt.Println(v1, v2)
```

3.4. Constantes

Los contrastes se definen como variables y almacenan datos pero los constantes el valor asignado no se modifican, los constantes se puede definir con un tipo de dato o también sin un tipo de dato solo se le asigna un valor y de acuerdo a su valor se convierte en tipo de constante.

```
package main
import "fmt"
//Definir una constante
const s string = "constante"
func main()  {
     const n = 25 //Constante de tipo entero
     fmt.Println(s , n)
}
```

Palabras reservadas de Go

Las palabras reservadas de Go, no puede definir como nombre de una variable, constante, función, array entro otros. Tienes que tener mucho cuidado al momento de definir los identificadores, las palabras reservados de Go son los siguientes.

break	default	funct	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

3.5. Salida por Pantalla

➔ Para mostrar datos por pantalla podemos usar varios métodos del paquete fmt, y dentro también podemos aplicar los caracteres especiales. como generar un espacio de tabulador y realizar un salto de línea, ya el método Print de fmt no hace salto de línea.

```
// Salida por pantalle
 fmt.Print("Hola\tMundo\n")
```

➔ También podemos concatenar con moma los valores literales y variables para mostrar diferentes tipos de datos.

```
//Concatenar datos
 nombre := "Alex Roel"
 edad := 25
 pi := 3.141592
 fmt.Println("Nombre: ", nombre, "Edad: ", edad, "\nValor de Pi: ", pi)
```

➔ Otro forma de mostrar estos datos es usando Printf que lo que hace es formatear nuestros datos y usando otros caracteres especiales:

- %s para mostrar datos de tipo string
- %d para mostrar datos de tipo entero
- %f para mostrar datos de tipo flotante

```
nombre := "Alex Roel"
edad := 25
pi := 3.141592
```

```
//Formatear datos
fmt.Printf("Nombre: %s Edad: %d \nValor de Pi: %f", nombre, edad, pi)
```

3.6. Entrada por Teclado

Para ingresar datos desde teclado y interactuar también usaremos los métodos del paquete fmt, como el Scanln.

```
var (
   nombre string
   edad int
   pi float64
 )
 //Entrada por teclado
 fmt.Print("Ingrese su Nombre: ")
 fmt.Scanln(&nombre) //Lee datos y guardar en la variable

 fmt.Print("Ingrese su Edad: ")
 fmt.Scanln(&edad)

 fmt.Print("Ingrese el valor de PI: ")
 fmt.Scanln(&pi)

 //Salida
 fmt.Printf("Nombre: %s Edad: %d \nValor de Pi: %f", nombre, edad, pi)
```

3.7. Operadores Aritméticos

Con operadores aritméticos podemos realizar operaciones matemáticas ya sean con valores literales o con variables. a = 10 y b = 5

Operador	Descripción	Ejemplo
+	Sumar	a + b resulta 15
–	Restar	a – b resulta 5
*	Multiplicar	a * b resulta 50
/	Dividir	a / b resulta 2
%	Calcular módulo de la división	a % b resulta 0

- Para esto podemos hacer un sistema donde ingreses dos números enteros y realizar la suma, resta, multiplicación, división y sacar el módulo de la división.

```go
package main

import "fmt"

func main()  {
 //Operadores Aritméticos
 var a, b int

 //Entrada
 fmt.Print("Ingrese número 01: ")
 fmt.Scanln(&a)

 fmt.Print("Ingrese número 02: ")
 fmt.Scanln(&b)

 //Proceso
 suma := a + b
 resta := a - b
 multi := a * b
 divi := a / b
 mod := a % b

 //Salida
 fmt.Println("La Suma es: ", suma)
 fmt.Println("La Resta es: ", resta)
```

```
fmt.Println("La Multiplicación es: ", multi)
fmt.Println("La División es: ", divi)
fmt.Println("El Módulo es: ", mod)
}
```

3.8. Operadores Relacionales

Los operadores relacionales lo que hace es comparar dos valores y devuelve un valor booleano de acuerdo a la comparación si es verda o falso. a = 1 y b = 2

Operador	Descripción	Ejemplo
==	Igualdad	a == b devuelve falso
!=	Distintos	a != b devuelve verdadero
>	Mayor que	a > b devuelve falso
<	Menor que	a < b devuelve falso
>=	Mayor o igual que	a >= b devuelve falso
<=	Menor o Igual que	a <= b devuelve verdadero

- Para esto podemos hacer un sistemas donde pide que ingresemos dos datos tipo entero y compara ese valores.

```
package main
import "fmt"
func main() {
   //Operadores Relacionales
   var a, b int
   //Entrada
   fmt.Print("Ingrese valor de a: ")
   fmt.Scanln(&a)
   fmt.Print("Ingrese valor de b: ")
   fmt.Scanln(&b)
   //Igualdad ==
```

```
    fmt.Println("¿Son Iguales? =>", a == b)
    //Distintos !=
    fmt.Println("¿Son Distintos? =>", a != b)
    //Menor que <
    fmt.Println("El Primero es menor que el segundo =>", a < b)
    //Menor o Igual que  <=
    fmt.Println("El Primero es menor o igual que el segundo =>", a <= b)
    //Mayor que >
    fmt.Println("El Primero es mayor que el segundo =>", a > b)
    //Mayor o Igual que >=
    fmt.Println("El Primero es mayor o igual que el segundo =>", a >= b)
}
```

3.9. Condicionales

Las condiciones son estructuras selectivas que de acuerdo al condición se ejecuta o no se ejecuta un bloque de código, entonces si el resultado de la condición es verdad entonces se ejecuta el bloque de código dentro de if, sino no se cumple la condición se ejecuta lo que está dentro de else.

```
//Condicionales
    if true {
        fmt.Println("Se Cumple la Condición")
    }else{
        fmt.Println("No se cumple la Condición")
    }
```

➔ Ejercicio: Crea un sistema que pida que ingrese un numero entero y el sistema detecte si es par o impar si es cero que diga que es neutro.

```
package main

import "fmt"

func main() {

    //Ejercicio
    var n int
    //Entrada
    fmt.Print("Ingrese un Número: ")
    fmt.Scanln(&n)
```

```
    if n == 0 {
        fmt.Println("ES NEUTRO")
    } else if n%2 == 0 {
        fmt.Println("ES PAR")
    } else {
        fmt.Println("ES IMPAR")
    }
}
```

3.10. Operadores Lógicos

Los operadores lógicos lo que hace es comparar dos valores booleanos, tiene tres operadores lógicos.

- **NOT (!):** lo que hace negar su valor. si su valor es true lo negara false

```
//Negación
 fmt.Prinln(! true)
```

- **AND(&&):** lo que hace es comparar dos valores booleanos y devuelve otro valor booleano, solo cuando ambos son true devuelve true.

```
//AND
fmt.Prinln(true && true)
```

- **OR(||):**Lo que hace es también comparar dos valores booleanos y devuelve otro valor booleano en este caso ambos o al meno uno tiene que ser true para que devuelva true.

```
//OR
   fmt.Prinln(true || false)
```

3.11. Casos

Los casos trabajan dentro de de instrucción swich, son estructura selectiva múltiple donde puede realizar varios casos. para ver este tema realizaremos un sistema.

- Ejercicio: Realizar un sistema que pida ingresar un número de 1 a 5 y devuelva escrito como por ejemplo ingrese un número que se 3 y devuelve TRES.

```
package main
```

```go
import "fmt"

func main() {
 var (
   n      int
   salida string
 )
 fmt.Print("Ingrese un numero de 1 a 5: ")
 fmt.Scanln(&n)

 // Switch y Case
 switch n {
 case 1:
   salida = "UNO"
 case 2:
   salida = "DOS"
 case 3:
   salida = "TRES"
 case 4:
   salida = "CUATRO"
 case 5:
   salida = "CINCO"
 default:
   salida = "NO ESTÁ ENTRE 1 Y 5"
 }

 //Salida
 fmt.Println(n, salida)
}
```

3.12. Operadores en Asignación

Los operadores en asignación lo que hacen es simplificar los operadores aritméticos en casos especiales. a = 5

Operador	Descripción	Ejemplo
=	Asignar un valor	a = 5

+=	Suma en Asignación	a = a + 3 => a += 3
-=	Resta en Asignación	a -= 3
*=	Multiplicación en asignación	a *= 3
/=	División en Asignación	a /= 3
%=	Módulo en asignación	a %= 3

```
//Operadores en asignación
   a := 2
   //a = a + 3
   a += 2
   fmt.Println(a)
```

Operadores de incremento y decremento

- Operador de incremento (++) incrementa su valor en 1.

```
//Operadores de Incremento
   b := 1
   b++
   fmt.Println(b)
```

- Operadores de decremento (--) decrementa en 1.

```
//Operadores de Decremento
   b := 2
   b--
   fmt.Println(b)
```

3.13. Bucle for

En Go solo existe for que puede utilizar como quieras. lo que hace es ejecutar varias veces un bloque de código.

```
//For cómo while
   c := 0
```

```
for c <= 5 {
    fmt.Println(c)
    c++
}
// Tipico for
for i := 0; i <= 5; i++ {
    fmt.Println(i)
}
```

3.14. Instrucción Break y Continue

- Break: esta instrucción puede aplicar dentro de un bucle para terminar con la ejecución.

```
//break
   for i := 0; i <= 10; i++ {
       if i == 5{
           fmt.Println("Se rompe el bucle")
           break
       }
       fmt.Println(i)
   }
```

- Continue: Esta instrucción termina con la ejecución pero no con el bucle.

```
//continue
   for i := 0; i <= 10; i++ {
       if i == 5{
           fmt.Println("Estamos en 5 ejecución")
           continue
           fmt.Println("No me ejecutaré")
       }
       fmt.Println(i)
   }
```

3.15. Ejercicio Propuesto

1. Crea un sistema que pueda adivinar si es Vocal o No. Para eso una letra tiene que ser ingresada por teclado.

2. Solicita un número e imprime todos los números pares e impares desde 1 hasta ese número con el mensaje "es par" o "es impar" si el número es 5 el resultado será: 1 - es impar 2 - es par 3 - es impar 4 - es par 5 - es impar

3. Escriba un programa que pida un número entero mayor que cero y calcule su factorial. La factorial es el resultado de multiplicar ese número por sus anteriores hasta la unidad.

4. Escribe un programa que permita ir introduciendo una serie indeterminada de números mientras su suma no supere 50. Cuando esto ocurra, se debe mostrar el total acumulado y el contador de cuantos números se han introducido

5. Escribe un programa con un bucle infinito con opciones para elegir, que pueda calcular el área de 2 figuras geométricas, triángulo y rectángulo. En primer lugar, pregunta de qué figura se quiere calcular el área, después solicita los datos que necesites para calcularlo.

    1) triángulo = b * h/2
    2) rectángulo = b * h
    3) Salir: Solo cuando escojas esta opción el bucle se detendrá

3.16. Ejercicio Resuelto #01

Crea un sistema que pueda adivinar si es Vocal o No. Para eso una letra tiene que ser ingresada por teclado.

```
package main
import "fmt"
func main () {
    //Variables
    var c , salida string

    //Entrada
    fmt.Print("Ingrese un Letra: ")
    fmt.Scanln(&c)

    //Proceso
    if (c == "a" || c == "A") {
        salida = "ES VOCAL"
    }else if (c == "e" || c == "E") {
        salida = "ES VOCAL"
    }else if (c == "i" || c == "I") {
        salida = "ES VOCAL"
    }else if (c == "o" || c == "O") {
        salida = "ES VOCAL"
```

```
    }else if (c == "u" || c == "U") {
        salida = "ES VOCAL"
    }else{
        salida = "NO ES VOCAL"
    }
    //Salida
    fmt.Println(salida)
}
```

3.17. Ejercicio Resuelto #02

Solicita un número e imprime todos los números pares e impares desde 1 hasta ese número con el mensaje "es par" o "es impar" si el número es 5 el resultado será: 1 - es impar 2 - es par 3 - es impar 4 - es par 5 - es impar

```
package main
import "fmt"

func main ()  {
  //Variable
  var n int

  //Entrada
  fmt.Println("Ingrese un número: ")
  fmt.Scanln(&n)

  //Proceso
  for i := 1; i <= n; i ++ {
    if i % 2 == 0 {
      fmt.Println(i, "=> ES PAR")
    } else {
      fmt.Println(i, "=> ES IMPAR")
    }
  }
}
```

3.18. Ejercicio Resuelto #03

Escriba un programa que pida un número entero mayor que cero y calcule su factorial. La factorial es el resultado de multiplicar ese número por sus anteriores hasta la unidad.

```go
package main
import "fmt"

func main ()  {
  //Variable
  var (
    n int
  )

  //Entrada
  fmt.Print("Ingrese un número: ")
  fmt.Scanln(&n)
  f := n

  //Proceso
  for i := 1; i < n; i ++ {
    f *= i
    fmt.Printf(f)
  }
}
```

3.19. Ejercicio Resuelto #04

Escribe un programa que permita ir introduciendo una serie indeterminada de números mientras su suma no supere 50. Cuando esto ocurra, se debe mostrar el total acumulado y el contador de cuantos números se han introducido.

```go
package main
import "fmt"

func main ()  {
  //Variable
  var n , c, suma int

  //Proceso
  for suma <= 50 {
    fmt.Print("Ingrese un número: ")
    fmt.Scanln(&n)
    c++
    suma += n
```

```
  }
  //Salida
  fmt.Println("Cantidad de Números", c)
  fmt.Println("Suma Total: ",suma)
}
```

3.20. Ejercicio Resuelto #05

Escribe un programa con un bucle infinito con opciones para elegir, que pueda calcular el área de 2 figuras geométricas, triángulo y rectángulo. En primer lugar, pregunta de qué figura se quiere calcular el área, después solicita los datos que necesites para calcularlo.

1) triángulo = b * h/2
2) rectángulo = b * h
3) Salir: Solo cuando escojas esta opción el bucle se detendrá

```
package main
import "fmt"

func main ()  {
  //Variable
  var opcion int

  //Proceso
  for true {
    fmt.Println("ELIGE UNA OPCIÓN - CALCULAR EL AREA - SALIR")
    fmt.Println("1) Triángulo \n2)Rectángulo \n3)Salir")
    fmt.Print("Ingrese un Opción: ")
    fmt.Scanln(&opcion)

    //Opción 1 Resolver el area del Triangulo
    if opcion == 1 {
      var b, h int
      fmt.Print("Ingrese la Base: ")
      fmt.Scanln(&b)
      fmt.Print("Ingrese la Altura: ")
      fmt.Scanln(&h)

      area := (b * h)/2
      fmt.Println("El Area del Triangulo es: ", area)
```

```
//Opción 2 Resolver el area del Rectangulo
    } else if opcion == 2 {
      var b, h int
      fmt.Print("Ingrese la Base: ")
      fmt.Scanln(&b)
      fmt.Print("Ingrese la Altura: ")
      fmt.Scanln(&h)

      area := b * h
      fmt.Println("El Area del Rectángulo es: ", area)
//Opción 3 Terminar el bucle infinito
    } else if opcion == 3 {
      fmt.Println("Cerrando sistema")
      break
    //Opción incorrecta
    } else {
      fmt.Println("Opcion incorrecta")
    }
  }
}
```

4. NIVEL INTERMEDIO – Manejo de Datos

4.1. Array

Array es un almacenador de datos fijos, puede almacenar cantidad de datos pero indicando cuantos datos va almacenar ese array, y en Go como es tipado solo puede almacenar un tipo de datos.

- Un array se define como las variables solo que en array se coloca dentro de los corchetes la cantidad de datos que va almcenar y para agregar y modificar su datos usamos el índice de cada dato que se coloca entre corchetes.

```
//Definición de un array vacío
   var array1 [2]string
   //Agregar datos a array vacío usando indice
   array1[0] = "dato01"
   array1[1] = "dato02"
   fmt.Println(array1)
```

- También podemos definir un array con valores ya asignados esto colocamos entre llaves.

```
//Array con valores
   array2 := [3]int{1, 2, 3}
   fmt.Println(array2)
   //Modificar un dato
   array2[2] = 4
   fmt.Println(array2)
```

4.2. Slicen

Los slicen son parecidas a los array que también almacena datos pero los slicen almacena cantidad de datos indeterminados, que puede almacenar cantidad de variables que tu desees.

- puedes definir un slicen vacío y luego agregar los datos con la función append.

```
//Definición de un slicen vacío
   var slicen1 []string

   //Agregar datos
   slicen1 = append(slicen1, "dato01", "dato02", "3")

   fmt.Println(slicen1)
```

4.3. Practica de Array y Slicen

Crea 1 arrays numeros y dos Slicen. El primero tendrá 5 números y el segundo se llamará pares y el tercero impares, ambos estarán vacíos. Después multiplica cada uno de los números del primer array por un número aleatorio entre 1 y 10, si el resultado es par guarda ese número en el Slicen de pares y si es impar en el Slicen de impares. Muestra por consola: -la multiplicación que se produce junto con su resultado con el formato 2 x 3 = 6 -el Slicen de pares e impares.

```
package main

import "fmt"

func main() {
    //Crear un array con cinco números
    numeros := [5]int{1, 2, 3, 4, 5}
    //Crear los Slicen vacíos pares y impares
    pares := []int{}
```

```
    impares := []int{}
    //Crear una variable para obtener el resultado
    m := 0

    //Crear un for para ejecutar 5 veces
    for i := 0; i < 5; i++ {
        //ingresar el número aleatorio
        var aleatorio int
        fmt.Print("Ingrese un número entre 1 y 10: ")
        fmt.Scanln(&aleatorio)
        //Realizar la multiplicación
        m = numeros[i] * aleatorio
        //Mostrar cómo se está multiplicando
        fmt.Println(numeros[i], "x", aleatorio, "=", m)

        //Compar si es par o no
        if m%2 == 0 {
            pares = append(pares, m)
        } else {
            impares = append(impares, m)
        }
    }

    //Mostrar los Slicen pares e impares
    fmt.Println("Pares ==> ", pares)
    fmt.Println("Impares ==> ", impares)
}
```

4.4. Funciones

Un funcio es útil para ordenar nuestros codigo, y tampier para utilizar un código múltiples veces.

- **Definición de una Función:**
  - Podemos crear otra función para ya no trabajar solo dentro de una función y esta funcion lo unico que va hacer es mostrar un saludo.

```
package main
```

```go
import "fmt"

func main() {
   //llamar la función Saludar para Ejecutar
   saludar()
}

//Definir una nueva Función
func saludar() {
   fmt.Println("Hola saludo de la Función Saludar")
   var nombre string
   fmt.Print("Nombre: ")
   fmt.Scanln(&nombre)
   fmt.Printf("Hola %s te saludo desde la función saludar \n", nombre)
}
```

- **Función con Retorno**:
  - Una función puede retornar un valor o múltiples valores y sea de un tipo o de diferentes tipos de datos, para recibir esos datos podemos guardar en una variable.

```go
package main
import "fmt"

func main() {
   //Guardar los datos de función datos
   nombre, edad := datos()

   fmt.Println(nombre, edad)
}

//Función con Retorno de Valores
func datos() (string, int) {
   var (
        nombre string
        edad   int
   )
   fmt.Print("Nombre: ")
   fmt.Scanln(&nombre)
   fmt.Print("Edad: ")
```

```
    fmt.Scanln(&edad)

    return nombre, edad
}
```

- Parámetros y Argumentos:
    - Los parámetros son valores que recibimos en la función y los argumentos son valores que enviamos a la función.

```
package main
import "fmt"

func main() {

    var a, b int
    fmt.Print("Ingrese n01: ")
    fmt.Scanln(&a)
    fmt.Print("Ingrese n02: ")
    fmt.Scanln(&b)

    //Enviar Argumentos
    suma := sumar(a, b)

    fmt.Println("La Suma es: ", suma)
}
//Funcion Con parametros
func sumar(a int, b int) int {
    suma := a + b
    return suma
}
```

4.5. Operaciones con Cadenas

Para hacer operaciones con cadena de caracteres tenemos que importar otra libreria la libreria string.

Documentación de paquetes de Strings:
https://golang.org/pkg/strings/?m=all

```
import (
    "fmt"
    "strings"
```

```
)
```

➔ Una operación que podemos hacer es convertir todo a mayúscula

```
   texto := "Hola Mundo, Hola Golang"
   //Convertir a Mayúscula
   fmt.Println(strings.ToUpper(texto))
```

➔ Convertir todo a minúscula

```
//Convertir a minuscula
   fmt.Println(strings.ToLower(texto))
```

➔ Modificar datos indicados - por ejemplo “Hola” modificar por “Hello”

```
//Modificar datos indicados
   fmt.Println(strings.Replace(texto, "Hola", "Hello", 1))
```

➔ Modificar todo los datos indicados - por ejemplo todo los datos que sean “Hola” por “Hello”

```
//Modificar todo los datos indicados
   fmt.Println(strings.ReplaceAll(texto, "Hola", "Hello"))
```

➔ Convertir Cadena de texto a un array

```
//Convertir a un array
   fmt.Println(strings.Split(texto, " "
```

➔ Convertir cadena de texto a Slicen

```
//Convertir a un slicen
   var slicen1 []string
   slicen1 = append(strings.Split(texto, ","))
   fmt.Println(slicen1, len(slicen1))
```

4.6. Paquetes

Con los paquetes podemos organizar más nuestro proyecto, un parque en Go sirve para contener nuestros archivos Go y organizar. Para crear un paquete simplemente creas una carpeta, pero a si sola no indica nada para que sea un paquete tenemos que crear un archivo de Go y indicar dentro del archivo el nombre del paquete (nombre de la carpeta creada para paquete).

```
mensaje //paquete (carpeta creada para paquete)
   saludo1.go //Archivó dentro de paquete
   saludo2.go
main.go //Archivo principal que trabajo con paquete main
```

Dentro de estos archivos tenemos que definir el paque y crear funciones. Para crear funciones podemos hacer de dos formas.

- **Privada**: En Go para crear un funcion o metodo privado simplemente iniciamos el nombre de la función con minúscula y esta función no puede ser accedida de fuera del paquete.
- **Pública**: En Go para crear un funcion o metodo publico solo inicias el nombre de la función con mayúscula y este funcione si puede ser accedida de fuera del paquete. Una función pública tiene que ser documentada iniciando con su nombre.

```go
package mensaje // Definir el paquete
import "fmt"
//funcionPrivada es una función privada
func funcionPrivada() {
    fmt.Println("Hola Soy una función Privada")
}
//FuncioPublica es una función Pública
func FuncioPublica() {
    fmt.Println("Hola Soy una función Pública")
    //Ejecutar la función Privada
    funcionPrivada()
}
```

Otro archivo para acceder a otras función ya se privadas o públicas de una función pero desde otro archivo dentro del paquete.

```
package mensaje // Definir el paquete
import "fmt"

//OtraFuncion Ejecuta una función Privada
func OtraFuncion() {
    fmt.Println("Ejecutar la función Privada desde Otra Función")
    funcionPrivada()
}
```

Después de crear el paquete *mensaje* y sus archivos de Go puedes importar en este caso desde el archivo *main* como el paquete *fmt*. En este caso con la dirección completa donde esta el paquete. Para acceder a sus funciones públicas podemos hacer como con paquete fmt.

```
package main
//Importar el paquete mensaje
import (
    "github.com/alexroel/gocurso/mensaje"
)
func main() {
    mensaje.FuncioPublica() //Acceder al FuncionPublica
    mensaje.OtraFuncion()
}
```

4.7. Práctica de Paquetes

Crea un nuevo paquete con nombre de aritmetica y dentro crea un archivo *operaciones.go* crear dos funciones una pública y otra privada. La pública tendrá de nombre `RealizarOperacion` dentro de esta función pedir que ingrese dos números y luego ese datos ingresados enviar mediante argumentos a la función sumar esta función será privada, que recibirá ese dos números mediante parámetros y retorna la suma. Desde la función main solo instanciar la función ReliazarOperaciones

```
package mensaje
import "fmt"
//RealizarOperacion esta función pide los datos y realiza la Operación
func RealizarOperacion() {
```

```
    var a, b int
    fmt.Print("Ingrese Número 01: ")
    fmt.Scanln(&a)
    fmt.Print("Ingrese Número 02: ")
    fmt.Scanln(&b)
    //Sumar
    fmt.Println("La suma es: ", sumar(a, b))
}
//Sumar realiza la suma
func sumar(a int, b int) int {
    return a + b
}
```

```
package main
//Importar el paquete mensaje
import (
    "github.com/alexroel/gocurso/mensaje"
)
func main() {
    mensaje.RealizarOperacion()//Acceder al RealizarOperacion
}
```

4.8. Mapa

Las mapas son como una listas que almacena cantidad de datos con clave y valor, Map en Go se parece a los diccionarios de Python.
Para definir una mapa se hace como variables o array, en este caso ponemos map luego dentro de los corchetes ponemos el tipo de dato que será la clave y luego el tipo de dato que será su valor y podemos hacer inicializando con valor predeterminado dentro de las llaves, clave y valor separados con dos puntos y cada elemento separados con coma.

```
//Definir un Map Con valor Iniciado
    mapa1 := map[string]int{
        "uno":  1,
        "dos":  2,
        "tres": 3,
    }
    fmt.Println(mapa1)
```

Para agregar o modificar datos solo llamar el nombre de map dentro de corchetes pones la clave o llamas por clave el valor que quieres modificar.

```
//Agregar y modificar datos más datos en map
    mapa1["cuatro"] = 4
    mapa1["cinco"] = 5

    fmt.Println(mapa1)
```

Para mostrar datos buscado por clave.

```
//Mostrar solo un dato buscando por clave
    fmt.Println(mapa1["cinco"])
```

Para eliminar un elemento de la mapa usas el método delete y dentro pones el nombre de la mapa más el clave para eliminar el elemento.

```
//Eliminar datos
    delete(mapa1, "cinco")

    fmt.Println(mapa1)
```

Para definir una mapa vacía usamos el metodo make para ahorrar una tarea extra.

```
//Definir un Map vacío
    mapa2 := make(map[int]string)

    mapa2[1] = "Uno"
    mapa2[2] = "Dos"
    mapa2[3] = "Tres"
    fmt.Println(mapa2)
```

Mopa con clave y valor de un solo tipo

```
//Mapa clave y valor de string
    mapa3 := make(map[string]string)
    mapa3["red"] = "Rojo"
```

```
    mapa3["blue"] = "Azul"
    fmt.Println(mapa3)
```

4.9. Práctica de Map

Definir el siguiente Map:

```
semana := map[int]string{
        1: "Domingo",
        2: "Lunes",
        3: "Jueves",
        4: "Miércoles",
        5: "Martes",
        6: "Viernes",
        7: "Sábado",
    }
```

Realizado un Listado de la mapa semana usando for y mostra como una lista clave más valor, para saber la cantidad de datos de la mapa usar funcione len(). luego modifica los datos que están correctos tomando en cuenta que los días de la semana inician desde domingo. los datos para modificar tienes que ser ingresados por teclado.
y luego actualizar la lista.

```
package main
import "fmt"
func main() {

    semana := map[int]string{
        1: "Domingo",
        2: "Lunes",
        3: "Jueves",
        4: "Miércoles",
        5: "Martes",
        6: "Viernes",
        7: "Sábado",
    }
    //Listar Datos
    for i := 1; i <= len(semana); i++ {
        fmt.Println(i, semana[i])
```

```
    }

    //Corregir Los datos
    var (
        c int
        k int
        v string
    )
    fmt.Println("Corrigir Datos")
    fmt.Print("Cuntos datos quieres Modificar: ")
    fmt.Scanln(&c)

    for i := 1; i <= c; i++ {
        fmt.Print("Ingrse la Clave: ")
        fmt.Scanln(&k)

        fmt.Print("Ingrese el Valor: ")
        fmt.Scanln(&v)

        //Modificando datos
        semana[k] = v
    }
    //Actualiza la lista
    for i := 1; i <= len(semana); i++ {
        fmt.Println(i, semana[i])
    }
}
```

4.10. For y Range

La función range itera los elementos de la mayoría de estructura de datos como Array, Slicen, Map y otros. se aplica con for para iterar cada elemento de una estructura. para integrar los elementos de un array se define los variables dentro de for, para iterar solo los elementos sin los índices pones un _ para decir índice bacio.

```
    array1 := []int{1, 2, 3, 4}
    //Iterara un Array sin indice
    for _, num := range array1 {
```

```
        fmt.Println(num)
    }
```

Iterar un Array con índice para eso se crea un variable en el for y también otra variable para su valor.

```
     array1 := []int{1, 2, 3, 4}
    //Iterara un Array con índice
    for i, num := range array1 {
        fmt.Println(i, "==>", num)
    }
```

Utilizando range podemos sumar todo los números dentro de array.

```
// Sumar todo los valores de array
    suma := 0
    for _, num := range array1 {
        suma += num
    }

    fmt.Println("La Suma ==> ", suma)
```

Con range podemos interar de manera facil los elementos de un Map.

```
    //Interar los elementos de un Map
    colores := map[string]string{
        "yellow": "Amarillo",
        "red":   "Rojo",
        "blue":  "Azul",
    }

    for k, v := range colores {
        fmt.Println(k, "==>", v)
    }
```

4.11. Crear nuevo tipo de Dato

En Go podemos crear un nuevo tipo de dato para esto podemos utilizar la instrucción type para crear un nuevo tipo de dato.

```go
package main
import "fmt"

//crear una tipo de dato
type entero int
type cadena string

func main() {
    var edad entero
    edad = 45
    fmt.Println(num)

    var nombre cadena
    nombre = "Alex Roel"
    fmt.Println(nombre)
}
```

4.12. Estructura

En Go las estructuras se pasen a programación orientada a objetos ya que en go no hay eso. en con creamos una estructura y luego de esa estructura podemos crear nuevos como objetos. Para esto podemos crear una estructura Curso y desa estructura podemos crear curso1 y curso2 y así más cursos.

```go
package main
import "fmt"

func main() {
    //Crear un curso de la estructura Curso
    curso1:= Curso{
        nombre: "Go",
        url:    "GoCurso",
        habilidad: []string{"backend", "2"},
    }
    fmt.Println(cusro1)

    //Crear más curso de la estructura Curso
    curso2 := new(Curso) //Crear una nueva estructura vacía
    fmt.Println(cusro2)
    curso2.nombre = "Python"
    curso2.url = "PythonCurso"
```

```
    curso2.habilidad = []string{"Backend"}
    fmt.Println(cusro2)
}
//Curso Estructura para crear cursos
type Curso struct {
    nombre string
    url    string
    habilidad []string
}
```

4.13. Métodos

Para crear métodos se crea como una función solo que en este caso antes de poner el nombre pones un parámetro y dentro una instancia y nombre de la estructura y este método solo será accedida mediante la estructura.

```
package main

import "fmt"

func main() {
    //Crear un curso de la estructura Curso
    curso1 := Curso{
        nombre:   "Go",
        url:      "GoCurso",
        habilidad: []string{"backend", "2"},
    }
    //Acceder al método inscribirse y enviar un dato
    curso1.Inscribirse("Alex")
    fmt.Println(cusro1)

}

//Curso Estructura para crear cursos
type Curso struct {
    nombre   string
    url      string
```

```
    habilidad []string
}

//Inscribirse en un método de Curso
func (c Curso) Inscribirse(nombre string) {
    fmt.Printf("La Persona %s se ha Registrado al curso %s \n",
nombre, c.nombre)
}
```

4.14. Herencia de Estructuras

Desde otra estructura podemos heredar otra estructura como sus campos y sus métodos, simplemente llamas dentro de la estructura nombre de la estructura que quieres heredar.

```
//Carrera nueva estructura
type Carrera struct {
    nombreCarrera string
    duracion      int
    Curso         //Heredar Curso
}
```

Para acceder a los campos y métodos heredados podemos acceder desde la instancia de nueva estructura.

```
//Crear una carrera
    carrera1 := new(Carrera)
    carrera1.nombreCarrera = "Sistemas"
    carrera1.duracion = 5
    carrera1.nombre = "Lp1"
    carrera1.url = "SistemasLp1"
    carrera1.habilidad = []string{"Programación", "Redes"}
    carrera1.Inscribirse("Alex Roel")

    fmt.Println(carrera1)
```

4.15. Práctica de Estructura

Integrantes de una Selección de Fútbol

Entrenador

- Nombre
- Edad
- Federación

  Viajar

  DirigirPartido

Futbolista

- Nombre
- Edad
- Dorsal

  Viajar

  JugarPartido

Como Diseñar

Selección

- Nombre
- Edad

  Viajar

Entrenador

- Federación

  DirigirPartido

Futbolista

- Dorsal

  JugarPartido

seleccionfutbol

seleccion.go

```go
package seleccionfutbol

import "fmt"

//Seleccion es estructura principal
type Seleccion struct {
   Nombre string
   Edad   int
}
```

```go

//Viajar metodo de seleccion
func (s Seleccion) Viajar() {
   fmt.Println("La Selección de Fútbol Vieja")
}
```

entrenador.go

```go
package seleccionfutbol

import "fmt"

//Entrenador es la estructura
type Entrenador struct {
   Seleccion
   Federacion string
}

//DirigirPartido es un método
func (e Entrenador) DirigirPartido() {
   fmt.Printf("El entrenador %s Dirige Partido \n", e.Nombre)
}
```

futbolista.go

```go
package seleccionfutbol

import "fmt"

//Futbolista es estructura
type Futbolista struct {
   Seleccion
   Dorsal int
}

//JugarPrtido es un método
func (f Futbolista) JugarPrtido() {
   fmt.Printf("El Futbolista %s Juega un partido\n", f.Nombre)
}
```

main.go

```go
package main

import (
   "fmt"
   "github.com/alexroel/gocurso/seleccionfutbol"
)

func main() {

   entrenador := new(seleccionfutbol.Entrenador)
   entrenador.Nombre = "Alex"
   entrenador.Edad = 25
   entrenador.Federacion = "FPF"
   fmt.Println(entrenador)
   entrenador.DirigirPartido()

   jugador1 := new(seleccionfutbol.Futbolista)
   jugador1.Nombre = "Roel"
   jugador1.Edad = 24
   jugador1.Dorsal = 10
   fmt.Println(jugador1)
   jugador1.JugarPrtido()

}
```

5. CREAR APLICACION – Calculadora (Práctica)

5.1. Descripción de la Aplicación - CALCULADORA

- **CAICULADORA**
- Crear un programa para operaciones aritméticas básicas.
- Sumar, Restar, Dividir y Multiplicar
- Usuario inserte los datos a través de consola

- **Objetivos**
- Recibir datos desde la consola con otros paquetes
- Repaso de if y Switch - case

- Repaso Funciones
- Repaso Estructuras - Array y Slicen
- Manejo de Errores

5.2. Librería bufio y strconv

bufio: El paquete bufio implementa Entrasa / Salida datos. Envuelve un objeto io.Reader o io.Writer, creando otro objeto (Reader o Writer) que también implementa la interfaz pero proporciona almacenamiento en búfer y algo de ayuda para E / S textuales.

https://golang.org/pkg/bufio/
En este caso vamos leer datos desde consola.

```
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("La calculador")
    scanner.Scan()
    operacion := scanner.Text()
    fmt.Println(operacion)
```

strconv: El paquete strconv implementa conversiones hacia y desde representaciones de cadenas de tipos de datos básicos.

https://golang.org/pkg/strconv/

```
valores := strings.Split(operacion, "+")
operador1, _ := strconv.Atoi(valores[0])
operador2, _ := strconv.Atoi(valores[1])
fmt.Println(operador1 + operador2)
```

5.3. Implementar Suma

sumar.go

```
package operadores

import (
    "fmt"
    "strconv"
    "strings"
)
```

```go
//Sumar realiza la suma
func Sumar(operacion string) int {
    valores := strings.Split(operacion, "+")
    resultado := 0

    for i := 0; i < len(valores); i++ {
        num, error := strconv.Atoi(valores[i])

        if error != nil {
            fmt.Println(error)
            fmt.Println("Error: Tiene que ingresar un número entero")
            fmt.Println("o ¡¡Sólo debes realizar con un Operador!!")
        } else {
            resultado += num
        }
    }

    return resultado
}
```

5.4. Manejo de Error

main,go

```go
package main

import (
   "bufio"
   "fmt"
   "os"
   "strings"

   "github.com/alexroel/calculadora/operadores"
```

```
)

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println("La calculador")
    scanner.Scan()
    operacion := scanner.Text()

    resultado := 0

    if strings.Contains(operacion, "+") {
        resultado = operadores.Sumar(operacion)
    } else if strings.Contains(operacion, "-") {
        resultado = operadores.Restar(operacion)
    } else if strings.Contains(operacion, "*") {
        resultado = operadores.Multiplicar(operacion)
    } else if strings.Contains(operacion, "/") {
        resultado = operadores.Dividir(operacion)
    } else {
        fmt.Println("Error: El Operador está mal Ingresado, ")
        fmt.Println("o ¡¡Sólo debes realizar con un Operador!!")
        fmt.Println("o ¡¡Este Operador no esta implementado!!")
    }
    fmt.Println(resultado)
}
```

5.5. Resta, Multiplicacion y Division

operadores

restar.go

```
package operadores

import (
    "fmt"
    "strconv"
    "strings"
)
```

```go
//Restar realiza la suma
func Restar(operacion string) int {
    valores := strings.Split(operacion, "-")
    resultado := 0

    for i := 0; i < len(valores); i++ {
        num, error := strconv.Atoi(valores[i])

        if error != nil {
            fmt.Println(error)
            fmt.Println("Error: Tiene que ingresar un número entero")
            fmt.Println("o ¡¡Sólo debes realizar con un Operador!!")
        } else {
            if resultado == 0 {
                resultado = num
            } else {
                resultado -= num
            }
        }
    }
```

multiplicar.go

```go
package operadores

import (
    "fmt"
    "strconv"
    "strings"
)

//Multiplicar realiza la suma
func Multiplicar(operacion string) int {
    valores := strings.Split(operacion, "*")
    resultado := 0

    for i := 0; i < len(valores); i++ {
        num, error := strconv.Atoi(valores[i])
```

```
        if error != nil {
            fmt.Println(error)
            fmt.Println("Error: Tiene que ingresar un número entero")
            fmt.Println("o ¡¡Sólo debes realizar con un Operador!!")
        } else {
            if resultado == 0 {
                resultado = num
            } else {
                resultado *= num
            }
        }
    }
    return resultado
}
```

dividir.go

```
package operadores

import (
    "fmt"
    "strconv"
    "strings"
)

//Dividir realiza la suma
func Dividir(operacion string) int {
    valores := strings.Split(operacion, "/")
    resultado := 0

    for i := 0; i < len(valores); i++ {
        num, error := strconv.Atoi(valores[i])

        if error != nil {
            fmt.Println(error)
            fmt.Println("Error: Tiene que ingresar un número entero")
            fmt.Println("o ¡¡Sólo debes realizar con un Operador!!")
        } else {
            if resultado == 0 {
                resultado = num
```

```
            } else {
                resultado /= num
            }
        }
    }

    return resultado
}
```

5.6. Compilar Aplicación

Para compilar tu aplicación los paquetes y sus dependencias podemos usar `go build main.go` y generará un archivo nuevo con código binario esto puedes ejecutar en cualquier sistema.

Para compilar y instalar los paquetes y sus dependencias de tu aplicación puede simplemente poner `go install` y se ba generar dentro de carpeta bin código binario de tu aplicación y puedes ejecutar de cualquier parte.

6. NIVEL AVANZADO – Temas Avanzados

6.1. Función Variádicas

Funciones Variádicas pueden ser llamadas con una cantidad indefinidos de variables. Podemos definir como cualquier función sólo que en los parámetros vamos definir de qué tipos de datos va recibir, agregando antes del tipo de dato más tres puntos en los parámetros.

Para esto podemos definir una función que sume todo los números que enviemos.

```go
func sum(numeros ...int) {
    fmt.Print(numeros, " ")
    total := 0
    for _, num := range numeros {
        total += num
    }
    fmt.Println(total)
```

```
}
```

A esta función que acabamos de crear podemos enviar cuantos numeros que sean necesarias.

```
    sum(1, 2)
    sum(1, 2, 3, 4)
```

También podemos agrupar en los números en un Slicen y enviar el slicen pero indicando más tres puntos.

```
    numeros := []int{1, 2, 3, 4}
    sum(numeros...)
```

6.2. Función Recursivo

Las funciones recursivas las funciones que se ejecutan a sí misma y podemos hacer como un bucle con esta funciones.
Para ver una función recursiva vamos sacar el factorial de un número. Para esto vamos crear una función factorial que recibirá un número entero y retorna el resultado de la factorial que será un entero.

```
func factorial(n int) int {
    if n == 0 {
        return 1
    }

    f := n * factorial(n-1)
    return f
}
```

Para Ejecutar solo llamamos la función y le enviamos un número.

```
func main() {
    fmt.Println(factorial(5))
}
```

6.3. Función Anónimas

Con las funciones anónimas se puede formar cierres, osea desde una función podemos retornar otra función. Las funciones anónimas son útiles cuando quieres definir una función en la cual no necesariamente quieres ponerle un nombre.

La Función secuencia retorna otra función, una función anónima definida dentro de la función secuencia. La función que retornamos encierra al variable i y eso forma un cierre.

```
func secuencia() func() int {
    i := 0

    return func() int {
        i++
        return i
    }
}
```

Para ejecutar esta función anónima crear una variable y asignado la función secuencia y cada vez que ejecutemos esta variable se ejecutara la funcion anonima.

```
func main() {
    ejecutarfuncion := secuencia()
    fmt.Println(ejecutarfuncion())
    fmt.Println(ejecutarfuncion())
    fmt.Println(ejecutarfuncion())
}
```

6.4. Punteros

Los punteros nos permite pasar referencias de calores y datos entre las funciones y métodos de nuestro programa.
Veamos un ejercicio sin apuntadores los datos modificados no se modificarán.

```
package main
import "fmt"

func main() {
    a := 25
```

```
    fmt.Println(a)
    modificar(a)
    fmt.Println(a)
}
func modificar(a int) {
    a = 16
}
```

Si para usar punteros o apuntadores utilizamos el operador & (et) o (ampersand) para identificar la memoria de la variable.

```
func main() {
    a := 25
    fmt.Println(&a)
}
func modificar(a int) {
    a = 16
    fmt.Println(&a)
}
```

Las memorias son diferentes porque lo que hace Go es copiar, por eso la modificación hacemo a la copia, ahora para hacer la modificación vamor usar el asterisco.

```
package main
import "fmt"
func main() {
    a := 25
    fmt.Println(a)
    modificar(&a)
    fmt.Println(a)
}
func modificar(a *int) {
    *a = 16
}
```

6.5. Práctica de Apuntadores

En esta Práctica tenemos que crear un estructura de persona con nombre y edad luego modificar los datos y imprimer sus dato y datos actualizados.

```go
package main
import "fmt"

func main() {
    p1 := persona{
        Nombre: "Alex",
        Edad:   25,
    }
    fmt.Println(p1)
    fmt.Println(p1.Nombre, p1.Edad)
    n := "Roel"
    e := 26
    p1.modificarDatos(n, e)
    fmt.Println(p1.Nombre, p1.Edad)
}

type persona struct {
    Nombre string
    Edad   int
}
func (p *persona) modificarDatos(nombre string, edad int) {
    p.Nombre = nombre
    p.Edad = edad
}
```

6.6. Introducción al Problema de Interfaces

Un Proyecto sin interface.

```go
package main

import "fmt"

func main() {
    per1 := perro{}
    moverPerro(per1)

    pez1 := pez{}
    moverPez(pez1)
```

```
    pajaro1 := pajaro{}
    moverPajaro(pajaro1)

}

type perro struct{}
type pez struct{}
type pajaro struct{}

func (perro) caminar() string {
    return "So perro y camino"
}
func (pez) nadar() string {
    return "Soy Pez y nado"
}
func (pajaro) volar() string {
    return "Soy pájaro y vuelo"
}

func moverPerro(per perro) {
    fmt.Println(per.caminar())
}

func moverPez(pe pez) {
    fmt.Println(pe.nadar())
}

func moverPajaro(pa pajaro) {
    fmt.Println(pa.volar())
}
```

6.7. Interfaces

Implementar interface son colecciones nombradas de signaturas nombradas.

```go
package main
import "fmt"

func main() {
   per1 := perro{}
   moverAnimal(per1)

   pez1 := pez{}
   moverAnimal(pez1)

   pajaro1 := pajaro{}
   moverAnimal(pajaro1)

}

type animal interface {
   mover() string
}
type perro struct{}
type pez struct{}
type pajaro struct{}

func (perro) mover() string {
   return "So perro y camino"
}
func (pez) mover() string {
   return "Soy Pez y nado"
}
func (pajaro) mover() string {
   return "Soy pájaro y vuelo"
}

func moverAnimal(a animal) {
   fmt.Println(a.mover())
}
```

6.8. Práctica de Interfaces

En esta práctica vamos sacar area y perimetro de un cuadrado y círculo utilizando interfaces.

Cuadrado

área = ancho * altura
perimetro = 2*ancho + 2*altura

Circulo

área = pi * (radio * radio)
perimetro = 2*pi * radio

```
package main
import (
    "fmt"
    "math"
)

type geometrica interface {
    area() float64
    perimetro() float64
}
type cuadrado struct {
    ancho, altura float64
}
type circulo struct {
    radio float64
}

func (cua cuadrado) area() float64 {
    return cua.ancho * cua.altura
}
func (cua cuadrado) perimetro() float64 {
    return 2*cua.ancho + 2*cua.altura
}

func (cir circulo) area() float64 {
    return math.Pi * (cir.radio * cir.radio)
}

func (cir circulo) perimetro() float64 {
    return 2 * math.Pi * cir.radio
}

func medidas(g geometrica) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perimetro())
```

```
}
func main() {
    cuadrado1 := cuadrado{
        ancho:  3,
        altura: 4,
    }

    circulo1 := circulo{
        radio: 5,
    }

    medidas(cuadrado1)
    medidas(circulo1)
}
```

6.9. Imprimir contenido Web

Para realizar este ejercicio vamos usar un interface ya implementado en un paquete y usaremos alguno paquetes más.
package io : https://golang.org/pkg/io/
package net/http : https://golang.org/pkg/net/http/

```
package main
import (
    "fmt"
    "io"
    "net/http"
)
type escritorWeb struct{}

func (escritorWeb) Write(p []byte) (int, error) {
    fmt.Println(string(p))
    return len(p), nil
}
func main() {
    respuesta, error := http.Get("http://google.com")
    if error != nil {
        fmt.Println(error)
    }
    e := escritorWeb{}
```

```
    io.Copy(e, respuesta.Body)
}
```

6.10. Introducción al Problema de la Concurrencia

Un Proyecto sin concurrencia:

```
package main
import (
    "fmt"
    "net/http"
    "time"
)

func main() {
    inicio := time.Now()
    servidores := []string{
        "http://youtube.com",
        "http://udemy.com",
        "http://google.com",
        "http://facebook.com",
    }

    for _, servidor := range servidores {
        revisarServidor(servidor)
    }

    tiempoPaso := time.Since(inicio)

    fmt.Println("Tiempo de ejecución", tiempoPaso)
}
func revisarServidor(servidor string) {
    _, error := http.Get(servidor)

    if error != nil {
        fmt.Println(servidor, "No está disponible")
```

```
    } else {
        fmt.Println(servidor, "Esta funcionando normal")
    }
}
```

6.11. GoRoutines

```
    for _, servidor := range servidores {
        //Implementacion de GoRoutines
        go revisarServidor(servidor)
    }
```

6.12. Canales - Channel

Con los canales veré las ejecuciones de Go Routines ya no sabemos qué es lo que esta pasando con la ejecucion de GoRoutines.

```
package main
import (
    "fmt"
    "net/http"
    "time"
)

func main() {
    inicio := time.Now()
    canal1 := make(chan string)
    servidores := []string{
        "http://youtube.com",
        "http://udemy.com",
        "http://google.com",
        "http://facebook.com",
    }

    for _, servidor := range servidores {
        go revisarServidor(servidor, canal1)
    }
```

```
    for i := 0; i < len(servidores); i++ {
        fmt.Println(<-canal1)
    }
    tiempoPaso := time.Since(inicio)

    fmt.Println("Tiempo de ejecución", tiempoPaso)
}
func revisarServidor(servidor string, canal chan string) {
    _, error := http.Get(servidor)

    if error != nil {
        //fmt.Println(servidor, "No está disponible")
        canal <- servidor + "No está disponible"
    } else {
        //fmt.Println(servidor, "Esta funcionando normal")
        canal <- servidor + "Esta funcionando normal"
    }
}
```

7. PRÁCTICA – Crear un Servidor Web

7.1. Descripción de la Aplicación

7.2. Manejador de Rutas

router.go

```
package servidor

import (
    "fmt"
    "net/http"
)
```

```go
//Router maneja ratas
type Router struct {
    //Reglas para manejar rutas que nuestro servidor puede manejarlas
    rules map[string]http.HandlerFunc
}

//NewRouter retornar una ruta
func NewRouter() *Router {
    return &Router{
        //Crear una mapa vacía xq nuestro servidor esta inicia vacío
        rules: make(map[string]http.HandlerFunc),
    }
}

//ServeHTTP método para escribir el mensaje de la ruta
func (r *Router) ServeHTTP(w http.ResponseWriter, request
*http.Request) {
    fmt.Fprintf(w, "Hola Mundo")
}
```

7.3. Crear servidor

server.go

```go
package servidor

import "net/http"

//Server Estructura de un servidor
type Server struct {
    port   string  //Puerto del servidor
    router *Router //Rutas del servidor
}

//NewServer escuchar conexiones
func NewServer(port string) *Server { //Recibe puerto y retorna
```

```
servidor
    return &Server{
        port:   port,         //Puerto
        router: NewRouter(), //crear Nueva ruta
    }
}

//Listen escuchar al servidor y devolver error
func (s *Server) Listen() error {
    //Procesar urls
    http.Handle("/home", s.router)
    //Escuchar las peticiones y obtener si hay error
    error := http.ListenAndServe(s.port, nil)
    if error != nil {
        return error
    }
    return nil
}
```

main.go

```
package main

import "github.com/alexroel/servidorweb/servidor"

func main() {

    //Crear un Servidor y enviar puerto 3000
    server := servidor.NewServer(":3000")
    server.Listen()
}
```

7.4.